

Using c++0x and the STL

Pedro Montuenga

University of Illinois at Urbana-Champaign

montuen2@illinois.edu

SpinFest 2014, PHENIX

July 23, 2014



Introduction

- c++11/14 is the latest version of the c++ standard
- The latest version of gcc at rcas is 4.4.7, which supports c++0x.
- This talk will focus on new functionality that is available by simply adding:
AM_CXXFLAGS= -Wall -Werror **-std=c++0x** to your Makefile.am or equivalent.
- Many features of c++11 can be used via boost libraries. Although, this will not be the focus of this talk.
- If unspecified, all novel objects are part of **std** namespace.

auto keyword

Deduces data type from initialization:

```
int main()
{
    auto a = 1 + 2;      // int
    auto b = 1.2 + 1;   // double

    vector<pair<double, myClass>> vp;

    [...]

    // vector<pair<double, myClass>>::iterator
    auto it_vp = vp.begin();
}
```

Smart Pointers

Motivation

```
int someFunction() {
    // Allocate new object that must be deleted
    TGraph *pg = new TGraph(arguments);
    [...]
    if (...)// function must return early
    {
        delete pg;
        return 1;
    }
    [...]
    delete pg;
    return 0;
}
```

Smart Pointers manage memory for you.

Smart Pointers

Solution

```
int someFunction() {
    // New declaration
    unique_ptr<TGraph> pg(new TGraph(arguments));
    [...]
    if (...)// function must be exited early
    {
        // no need to delete
        return 1;
    }
    [...]
    // no need to delete
    return 0;
}
```

unique_ptr

- By far the most common smart pointer.
- It takes as much memory as a raw pointer.
- It deletes the object after it goes out of scope.

```
{  
    unique_ptr<TClass> pc(new TClass(arguments));  
    [...]  
    return 0;  
} // pc out of scope, TClass object deleted
```

- It ensures that one and only one smart pointer refers to TClass instance.

```
unique_ptr<TClass> pc(new TClass(arguments));  
unique_ptr<TClass> pc2 = pc; // compiler error!
```

- unique_ptr overloads operator*(), i.e., it can be dereferenced just like normal pointers.

shared_ptr

- Keeps a *reference count* of how many pointers are referencing the object.
- After count goes to zero, it deletes the object

```
{  
    shared_ptr<TClass> pc(new TClass(arguments));  
    {  
        // ref count increases by 1  
        shared_ptr<TClass> pc2 = pc;  
    } //pc2 destroyed, ref count decreases by 1  
} // ref count now zero, object destroyed
```

- The only choice when making vectors of pointers, as STL containers cannot hold unique_ptrs

Problem: What about custom delete?

```
int someFunction() {
    unique_ptr<TFile> pg(new TFile(arguments));
    [...]
    if (... ) // function must be exited early
    {
        // forgot to call pg->Close();
        return 1;
    }
    [...]
    // forgot to call pg->Close();
    return 0;
}
```

Custom Delete functors

Both `unique_ptr` and `shared_ptr` support custom delete functions

```
struct delFile1
{
    void operator()(TFile *pf) const
    {
        pf->Close();
        delete pf;
    }
};

struct delFile2
{
    void operator()(TFile *pf) const
    {
        pf->Close('r');
        delete pf;
    }
};

void someFunction() {
    unique_ptr<TFile, delFile1> upf1(new TFile(arguments));
    unique_ptr<TFile, delFile2> upf2(new TFile(arguments));

    shared_ptr<TFile> spf(new TFile(arguments), delFile1);
}
```

void functions are also possible, although functors give better performance and are easier to use

Smart Pointers

Summary

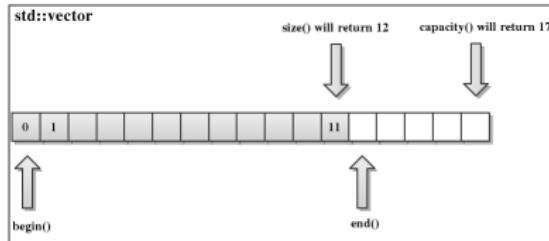
- When allocating objects dynamically (i.e., `new`), use smart pointers instead of raw pointers.
- Use `unique_ptr` for exclusive ownership.
- Use `shared_ptr` for shared ownership.
- For containers of pointers, only `shared_ptr` is legal.
- **Further Reading:** `weak_ptr`, RAII.

Algorithms with the STL

- The Standard Template Library (STL) is especially designed for high-performance algorithm implementation.
- It comprises:
 - 1 **Containers:** `vector`, `map`, `set`, etc.
 - 2 **Iterators:** Input, Output, Forward, Bidirectional, RandomAccess.
 - 3 **Algorithms:** sorting, partition, find, `for_each`, etc.
 - 4 **Functionals:** For algorithms with user-defined data-types.

Sequence Containers - std::vector

- std::vector is a **dynamically allocated** array.
- It holds a variable number of objects (`size()`)



create and share your own diagrams at gliffy.com



- When `size() == capacity()` a new array with $2 \times (\text{capacity}())$ is allocated and the old objects are copied to the new array. Making insertion *amortized constant time*.
- Other sequence containers include deque, stack, queue, string.

Vectors vs Arrays

- vectors resize when needed.
- vectors support begin(), end(), size() and capacity().
- vectors check bounds.

```
vector<int> v;           // [ ]  
v.push_back(5);          // [5]  
v.push_back(6);          // [5 6]  
v.at(0) = 4;             // [4 6]  
v.at(2) = 3;             // ERROR, failed range_check!
```

- Elements can be accessed via operator[].

```
cout << v[1] << endl; // OK, prints '6'  
cout << v[2] << endl; // ERROR, segmentation fault
```

- Easy to pass as arrays:

```
function(&v[0]);
```

Algorithms on vectors

- The STL supports both general and container-specific algorithm implementations with **performance guarantees**
- Examples:

```
vector<double> v;  
[...]  
// sorts in increasing order  
vector::sort(v.begin(), v.end());  
  
// returns pointer to first element from  
// left to right equivalent to 5.3  
find(v.begin(), v.end(), 5.3);
```

- On sorted arrays:

```
// whether element exists  
cout << binary_search(v.begin(), v.end()) << endl;  
// returns pair<vector<double>::iterator,  
//           vector<double>::iterator>  
// to first and last incidence  
auto bounds = equal_range(v.begin(), v.end(), 5.3);
```

Algorithms with classes

Motivation

```
class Point{  
private:  
    float x, y;  
public:  
    float GetX();  
    float GetY();  
};  
  
int main(){  
    vector<Point> vP;  
    [...]  
}
```

- What is the mean y value?
- What if we want to sort in increasing order in x?
- How about testing a predicate on Points such as distance to a given point?

std::bind

Example I: obtain mean y value

```
float addY(float x, Point p) {return x + p.GetY(); }

int main(){
    vector<Point> vP;
    [...]

    // get mean y value

    m_y = accumulate(vP.begin(), vP.end(), 0.0, addY)
          / vP.size();
}
```

Alternatively we can do it in one line without writing outside functions:

```
m_y = accumulate(vP.begin(), vP.end(), 0.0,
                  bind(plus<float>(), _1, bind(&Point::GetY, _2)))
          / vP.size();
```

std::bind

Introduction

- std::bind returns a function object, it can bind an arbitrary number of placeholders (_1, _2, _3, ...).

```
int Add(int a, int b);
int main() {
    auto fAdd = bind(&Add, _1, _2);
    cout << fAdd(3, 5) << endl; // prints out 8

    auto AddTwo = bind(&Add, 2, _1);

    cout << AddTwo(5) << endl; // prints out 7

    auto AddSevenAndTen = bind(&Add, 7, 10);

    cout << AddSevenAndTen() << endl; // 17
    [...]
}
```

std::bind

More Examples

- Sort in decreasing order of x:

```
sort(vP.begin(), vP.end(),
      bind(greater<int>(),
            bind(&Point::GetX, _1),
            bind(&Point::GetX, _2)
      )
    )
```

- Test whether points are more than distance d=5.0 away from (1.5,2.5)

```
float distance(float x0, float y0,
               float currX, float currY){
    return sqrt(pow(currX - x0, 2)
                + pow(currY - y0, 2));
};

partition(vP.begin(), vP.end(),
          bind(less<float>(),
                bind(distance, 1.5, 2.5, _1, _2), 5.0));
```

std::bind

More Examples

```
{  
    TH1F x_histo(arguments);  
    TH1F y_histo(arguments);  
  
    vector<Point> vP;  
    [...]  
    // In every iteration calls TH1F.Fill(x)  
    for_each(vP.begin(), vP.end(),  
              bind(&TH1F::Fill, x_histo,  
                    bind(&Point::GetX(), _1)));  
  
    // Get y distribution for the percentile 90 of x values  
  
    sort(vP.begin(), vP.end(),  
          bind(less<float>(),  
                bind(&Point::GetX, _1)  
                bind(&Point::GetX, _2)));  
  
    auto new_begin = vP.begin() + (vP.size() * 9 / 10);  
  
    for_each(new_begin, nP.end(),  
              bind(&TH1F::Fill, y_histo,  
                    bind(&Point::GetY(), _1)));  
}
```

STL Algorithms

Summary

- The STL offers close to optimal performance.
- Prefer STL algorithms to their for-loop counterparts

```
vector<int> v;
[...]
// find integer 2
auto first = find(v.begin(), v.end(), 2);

// using for-loop
vector<int>::iterator first;
for(auto it = v.begin(), it != v.end(), ++it){
    if(*it == 2){
        first = it;
        break;
    }
}
```

- Further reading: functionals, std::lambda.

Table of contents

- 1 Introduction
- 2 auto keyword
- 3 Smart Pointers
- 4 Algorithms with the STL
- 5 std::bind

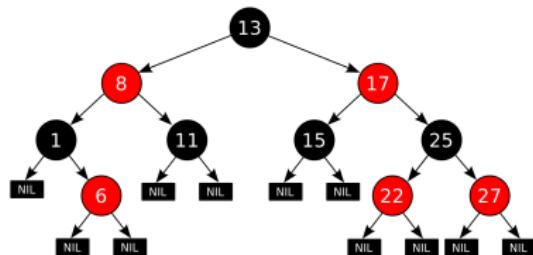
References:

- Effective C++
- Effective STL
- EFMC++
- Accelerated C++
- Other online resources

Thank you!

Associative Containers - std::map and std::set

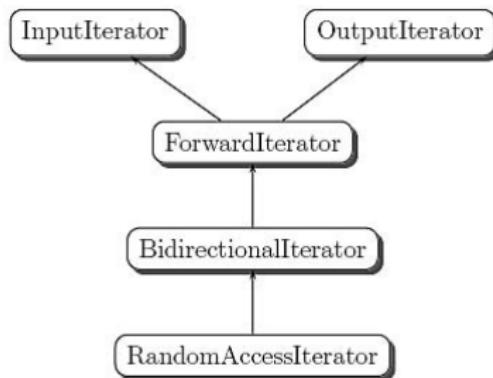
- std::map and std::set are permanently sorted containers.
- They are implemented in terms of a balanced (red-black) tree.



- std::map is a dictionary (key-value pair), std::set stores only values.
- Other containers include multiset, multimap, unordered_set, unordered_map, unordered_multiset, unordered_multimap.

Iterators

- Every STL container comes with its own iterator.
- All STL iterators have the capacity to:
 - 1 Move forward
 - 2 pointers to the first (`begin()`) and last (`end()`) iterator.
- Therefore, all STL containers can operate on every element between two given iterators.



STL Algorithms: